**1. “+”连接符**

**1.1 “+”连接符的实现原理**

Java语言为“+”连接符以及对象转换为字符串提供了特殊的支持，字符串对象可以使用“+”连接其他对象。其中字符串连接是通过 StringBuilder（或 StringBuffer）类及其append 方法实现的，对象转换为字符串是通过 toString 方法实现的，该方法由 Object 类定义，并可被 Java 中的所有类继承。有关字符连接和转换的更多信息，可以参阅 Gosling、Joy 和 Steele 合著的 《The Java Language Specification》

我们可以通过反编译验证一下

/\*\*

\* 测试代码

\*/

public class Test {

public static void main(String[] args) {

int i = 10;

String s = "abc";

System.out.println(s + i);

}

}

/\*\*

\* 反编译后

\*/

public class Test {

public static void main(String args[]) { //删除了默认构造函数和字节码

byte byte0 = 10;

String s = "abc";

System.out.println((new StringBuilder()).append(s).append(byte0).toString());

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21

由上可以看出，Java中使用"+"连接字符串对象时，会创建一个StringBuilder()对象，并调用append()方法将数据拼接，最后调用toString()方法返回拼接好的字符串。由于append()方法的各种重载形式会调用String.valueOf方法，所以我们可以认为：

//以下两者是等价的

s = i + ""

s = String.valueOf(i);

//以下两者也是等价的

s = "abc" + i;

s = new StringBuilder("abc").append(i).toString();

* 1
* 2
* 3
* 4
* 5
* 6
* 7

**1.2 “+”连接符的效率**

使用“+”连接符时，JVM会隐式创建StringBuilder对象，这种方式在大部分情况下并不会造成效率的损失，不过在进行大量循环拼接字符串时则需要注意。

String s = "abc";

for (int i=0; i<10000; i++) {

s += "abc";

}

/\*\*

\* 反编译后

\*/

String s = "abc";

for(int i = 0; i < 1000; i++) {

s = (new StringBuilder()).append(s).append("abc").toString();

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12

这样由于大量StringBuilder创建在堆内存中，肯定会造成效率的损失，所以在这种情况下建议在循环体外创建一个StringBuilder对象调用append()方法手动拼接（如上面例子如果使用手动拼接运行时间将缩小到1/200左右）。

/\*\*

\* 循环中使用StringBuilder代替“+”连接符

\*/

StringBuilder sb = new StringBuilder("abc");

for (int i = 0; i < 1000; i++) {

sb.append("abc");

}

sb.toString();

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8

与此之外还有一种特殊情况，也就是当"+"两端均为编译期确定的字符串常量时，编译器会进行相应的优化，直接将两个字符串常量拼接好，例如：

System.out.println("Hello" + "World");

/\*\*

\* 反编译后

\*/

System.out.println("HelloWorld");

* 1
* 2
* 3
* 4
* 5
* 6

/\*\*

\* 编译期确定

\* 对于final修饰的变量，它在编译时被解析为常量值的一个本地拷贝存储到自己的常量池中或嵌入到它的字节码流中。

\* 所以此时的"a" + s1和"a" + "b"效果是一样的。故结果为true。

\*/

String s0 = "ab";

final String s1 = "b";

String s2 = "a" + s1;

System.out.println((s0 == s2)); //result = true

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9

/\*\*

\* 编译期无法确定

\* 这里面虽然将s1用final修饰了，但是由于其赋值是通过方法调用返回的，那么它的值只能在运行期间确定

\* 因此s0和s2指向的不是同一个对象，故上面程序的结果为false。

\*/

String s0 = "ab";

final String s1 = getS1();

String s2 = "a" + s1;

System.out.println((s0 == s2)); //result = false

public String getS1() {

return "b";

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13

综上，“+”连接符对于直接相加的字符串常量效率很高，因为在编译期间便确定了它的值，也就是说形如"I"+“love”+“java”; 的字符串相加，在编译期间便被优化成了"Ilovejava"。对于间接相加（即包含字符串引用，且编译期无法确定值的），形如s1+s2+s3; 效率要比直接相加低，因为在编译器不会对引用变量进行优化。

**2. 字符串常量池**

在Java的内存分配中，总共3种常量池，分别是**Class常量池**、**运行时常量池**、**字符串常量池**。

字符串的分配和其他对象分配一样，是需要消耗高昂的时间和空间的，而且字符串使用的非常多。JVM为了提高性能和减少内存的开销，在实例化字符串的时候进行了一些优化：使用字符串常量池。每当创建字符串常量时，JVM会首先检查字符串常量池，如果该字符串已经存在常量池中，那么就直接返回常量池中的实例引用。如果字符串不存在常量池中，就会实例化该字符串并且将其放到常量池中。由于String字符串的不可变性，**常量池中一定不存在两个相同的字符串**。

/\*\*

\* 字符串常量池中的字符串只存在一份！

\* 运行结果为true

\*/

String s1 = "hello world!";

String s2 = "hello world!";

System.out.println(s1 == s2);

* 1
* 2
* 3
* 4
* 5
* 6
* 7

**2.1 内存区域**

在HotSpot VM中字符串常量池是通过一个StringTable类实现的，它是一个Hash表，默认值大小长度是1009；这个StringTable在每个HotSpot VM的实例中只有一份，被所有的类共享；字符串常量由一个一个字符组成，放在了StringTable上。要注意的是，如果放进String Pool的String非常多，就会造成Hash冲突严重，从而导致链表会很长，而链表长了后直接会造成的影响就是当调用String.intern时性能会大幅下降（因为要一个一个找）。

在JDK6及之前版本，字符串常量池是放在Perm Gen区(也就是方法区)中的，StringTable的长度是固定的1009；在JDK7版本中，字符串常量池被移到了堆中，StringTable的长度可以通过\*\*-XX:StringTableSize=66666\*\*参数指定。至于JDK7为什么把常量池移动到堆上实现，原因可能是由于方法区的内存空间太小且不方便扩展，而堆的内存空间比较大且扩展方便。

**2.2 存放的内容**

在JDK6及之前版本中，String Pool里放的都是字符串常量；在JDK7.0中，由于String.intern()发生了改变，因此String Pool中也可以存放放于堆内的字符串对象的引用。

/\*\*

\* 运行结果为true false

\*/

String s1 = "AB";

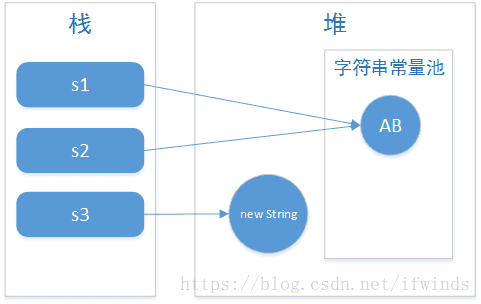
String s2 = "AB";

String s3 = new String("AB");

System.out.println(s1 == s2);

System.out.println(s1 == s3);

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8

由于常量池中不存在两个相同的对象，所以s1和s2都是指向JVM字符串常量池中的"AB"对象。new关键字一定会产生一个对象，并且这个对象存储在堆中。所以String s3 = new String(“AB”);产生了两个对象：保存在栈中的s3和保存堆中的String对象。  


当执行String s1 = "AB"时，JVM首先会去字符串常量池中检查是否存在"AB"对象，如果不存在，则在字符串常量池中创建"AB"对象，并将"AB"对象的地址返回给s1；如果存在，则不创建任何对象，直接将字符串常量池中"AB"对象的地址返回给s1。

**3. intern方法**

直接使用双引号声明出来的String对象会直接存储在字符串常量池中，如果不是用双引号声明的String对象，可以使用String提供的intern方法。intern 方法是一个native方法，intern方法会从字符串常量池中查询当前字符串是否存在，如果存在，就直接返回当前字符串；如果不存在就会将当前字符串放入常量池中，之后再返回。

JDK1.7的改动：

1. 将String常量池 从 Perm 区移动到了 Java Heap区
2. String.intern() 方法时，如果存在堆中的对象，会直接保存对象的引用，而不会重新创建对象。

/\*\*

\* Returns a canonical representation for the string object.

\* <p>

\* A pool of strings, initially empty, is maintained privately by the

\* class {@code String}.

\* <p>

\* When the intern method is invoked, if the pool already contains a

\* string equal to this {@code String} object as determined by

\* the {@link #equals(Object)} method, then the string from the pool is

\* returned. Otherwise, this {@code String} object is added to the

\* pool and a reference to this {@code String} object is returned.

\* <p>

\* It follows that for any two strings {@code s} and {@code t},

\* {@code s.intern() == t.intern()} is {@code true}

\* if and only if {@code s.equals(t)} is {@code true}.

\* <p>

\* All literal strings and string-valued constant expressions are

\* interned. String literals are defined in section 3.10.5 of the

\* <cite>The Java&trade; Language Specification</cite>.

\*

\* @return a string that has the same contents as this string, but is

\* guaranteed to be from a pool of unique strings.

\*/

public native String intern();

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24

**3.1 intern的用法**

static final int MAX = 1000 \* 10000;

static final String[] arr = new String[MAX];

public static void main(String[] args) throws Exception {

Integer[] DB\_DATA = new Integer[10];

Random random = new Random(10 \* 10000);

for (int i = 0; i < DB\_DATA.length; i++) {

DB\_DATA[i] = random.nextInt();

}

long t = System.currentTimeMillis();

for (int i = 0; i < MAX; i++) {

//arr[i] = new String(String.valueOf(DB\_DATA[i % DB\_DATA.length]));

arr[i] = new String(String.valueOf(DB\_DATA[i % DB\_DATA.length])).intern();

}

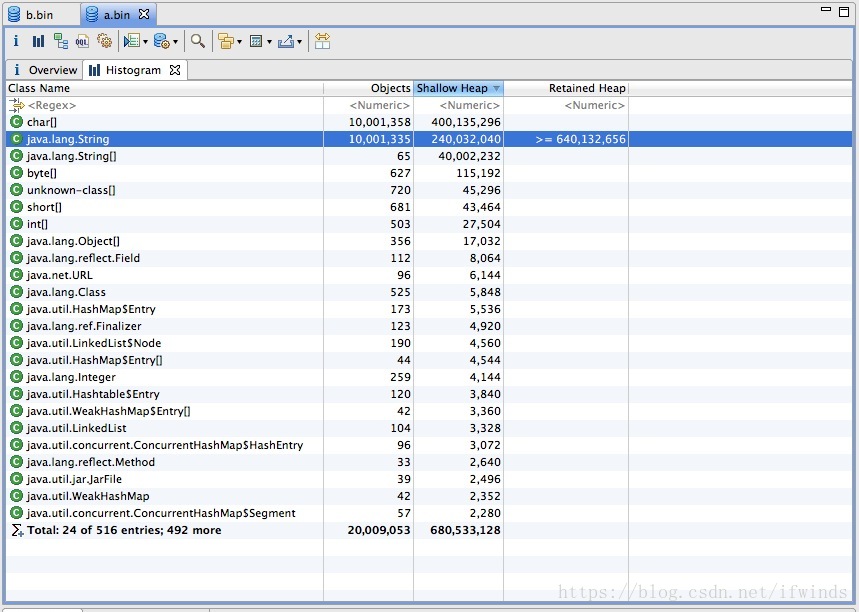
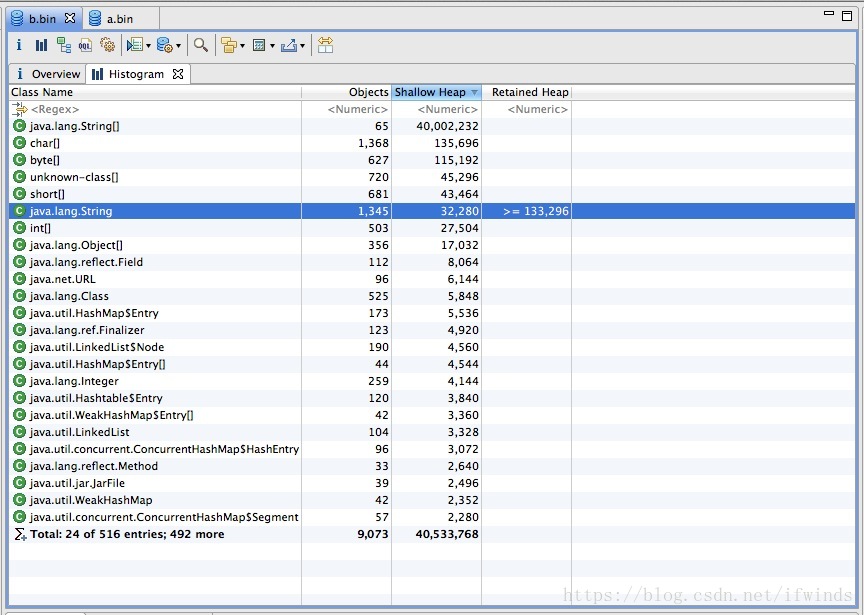
System.out.println((System.currentTimeMillis() - t) + "ms");

System.gc();

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18

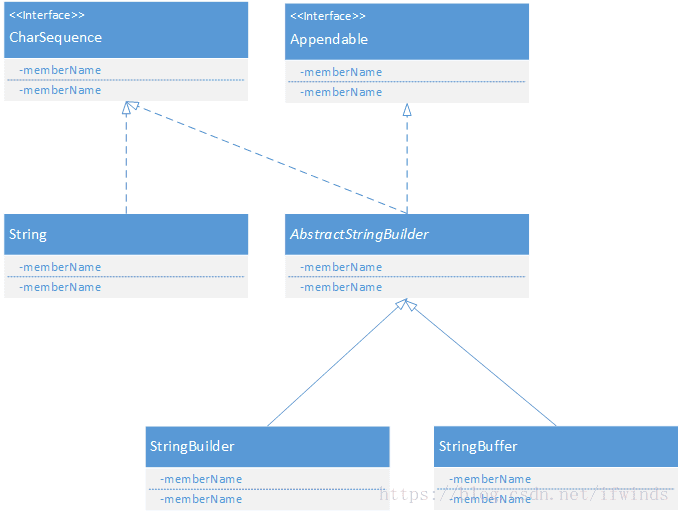
运行的参数是：-Xmx2g -Xms2g -Xmn1500M 上述代码是一个演示代码，其中有两条语句不一样，一条是未使用 intern，一条是使用 intern。结果如下图

未使用intern，耗时826ms：  
  
使用intern，耗时2160ms：  
  
通过上述结果，我们发现不使用 intern 的代码生成了1000w 个字符串，占用了大约640m 空间。 使用了 intern 的代码生成了1345个字符串，占用总空间 133k 左右。其实通过观察程序中只是用到了10个字符串，所以准确计算后应该是正好相差100w 倍。虽然例子有些极端，但确实能准确反应出 intern 使用后产生的巨大空间节省。

细心的同学会发现使用了 intern 方法后时间上有了一些增长。这是因为程序中每次都是用了 new String 后，然后又进行 intern 操作的耗时时间，这一点如果在内存空间充足的情况下确实是无法避免的，但我们平时使用时，内存空间肯定不是无限大的，不使用 intern 占用空间导致 jvm 垃圾回收的时间是要远远大于这点时间的。 毕竟这里使用了1000w次intern 才多出来1秒钟多的时间。

**4. String、StringBuilder和StringBuffer**

**4.1 继承结构**



**4.2 主要区别**

1）String是不可变字符序列，StringBuilder和StringBuffer是可变字符序列。  
2）执行速度StringBuilder > StringBuffer > String。  
3）StringBuilder是非线程安全的，StringBuffer是线程安全的。

**5. 总结**

String类是我们使用频率最高的类之一，也是面试官经常考察的题目，下面是一个小测验。

public static void main(String[] args) {

String s1 = "AB";

String s2 = new String("AB");

String s3 = "A";

String s4 = "B";

String s5 = "A" + "B";

String s6 = s3 + s4;

System.out.println(s1 == s2);

System.out.println(s1 == s5);

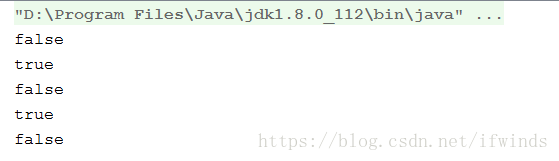
System.out.println(s1 == s6);

System.out.println(s1 == s6.intern());

System.out.println(s2 == s2.intern());

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13

运行结果：  


解析：真正理解此题目需要清楚以下三点  
1）直接使用双引号声明出来的String对象会直接存储在常量池中；  
2）String对象的intern方法会得到字符串对象在常量池中对应的引用，如果常量池中没有对应的字符串，则该字符串将被添加到常量池中，然后返回常量池中字符串的引用；  
3） 字符串的+操作其本质是创建了StringBuilder对象进行append操作，然后将拼接后的StringBuilder对象用toString方法处理成String对象，这一点可以用javap -c命令获得class文件对应的JVM字节码指令就可以看出来。  
